Module 02: Key Exchange and the TLS 1.3 Protocol

Week-04, Part I: Implementing The SIGMA Protocol

Jan Gilcher, Fernando Virdia, Kenny Paterson, and Lukas Burkhalter

jan.gilcher@inf.ethz.ch, fernando.virdia@inf.ethz.ch
kenny.paterson@inf.ethz.ch, lubu@inf.ethz.ch

October 2021

Welcome to the first lab for this module. This lab is part of the Module-02 on *Key Exchange and TLS 1.3*. Due to technical limitations the labs will only take place in person. You are also welcome to post your questions and to join ongoing discussions on the Moodle forum at the following link:

`https://moodle-app2.let.ethz.ch/mod/forum/view.php?id=616927`

Students who cannot attend the labs in person are especially encouraged to use Moodle to ask questions.

Overview

Welcome to the first lab of the module. This lab serves as an opportunity to learn how to combine the symmetric and asymmetric cryptographic primitives you will be implementing for the TLS lab assessment by implementing the SIGMA protocol – an "authenticated key exchange" (AKE) protocol. An AKE protocol enables two parties to end up with a shared secret key in a secure and authenticated manner. In other words, the security goals of an AKE protocol are two-fold. First, a computationally bounded adversary should learn no information about the value of the exchanged secret between the participating parties. Secondly, a computationally bounded adversary should not be able to impersonate one of the two participating parties in the protocol.

In this lab, you will be implementing a specific candidate AKE protocol that establishes a shared secret value between a pair of parties via Diffie-Hellman key exchange, while additionally authenticating the corresponding public keys and identities of the parties via digital signatures and message authentication codes. As you might have guessed by now (the title of the lab is a real giveaway), you will be implementing the SIGMA protocol. More concretely, you will be implementing a variant of the SIGMA protocol *that uses nonces* and has *identity hiding*: this variant of the SIGMA protocol is similar in structure to the TLS 1.3 Handshake Protocol which will be explained in the lectures next week. As explained concisely by Hugo Krawczyk in [1]:

> The SIGMA protocols introduce a general approach to building authenticated Diffie-Hellman protocols using a careful combination of digital signatures and a MAC (message authentication) function. We call this the **SIG**n-and-**MA**c" approach which is also the reason for the **SIGMA** acronym.

Getting Started

Please note that we expect you to use Python-3 (>=3.4) for this lab. We will not accept submissions/solutions coded in any other language, including older versions of Python. *So please make sure that your submissions are Python-3 (>=3.4) compatible.* For this lab you will need to install an elliptic-curve library using `pip3 install tinyec`, and modern AEAD libraries using PyCryptodome (`pip3 install pycryptodomex`). Similarly to the last module, we are also providing a virtual machine (VM) image that has the necessary dependencies pre-installed and ready to use.

The link to access and download the VM is available on Moodle. Below we provide you with a simple set of instructions on how to use this VM:

- Boot the VM using your favorite virtualization software. For example, you can download and use the Oracle VM Virtualbox from the url below:

  `https://www.virtualbox.org/`
- Login using the root password `isl`
- At this point, you can run any Python script(s) that are required for this assessment.

Note that when automatically evaluating your code, we will use the same build environment. So using the VM would additionally allow you to pre-test your submission in an environment resembling our automated testing environment.

Before we begin, we introduce the cryptographic background and notation that we will be using throughout this lab sheet.

Background and Notations

- $(x, X = g^x) \leftarrow_R \mathsf{DH.KeyGen}(\lambda)$: denotes the Diffie-Hellman key generation algorithm. It takes as input a security parameter λ, and outputs a secret/public Diffie-Hellman pair $(x, X = g^x)$.
- $(g^{xy}) \leftarrow \mathsf{DH.DH}(x, Y)$: denotes the Diffie-Hellman computation algorithm. It takes as input a Diffie-Hellman secret value x and a Diffie-Hellman public value Y, and outputs a shared secret g^{xy}.

  For all implementations in this lab, we will be using elliptic-curve Diffie-Hellman to perform DH operations, imported from **tinyec**. For greater readability and ease of exposition, we will be using the exponential notation for Diffie-Hellman throughout i.e. we write g^x in place of scalar multiplication $[x]P$ where P is a base point on an elliptic curve.
- $k' \leftarrow \mathsf{KDF}(r, s, c, L)$ is a key derivation function. It takes as input some randomness r, some (optional) salt s, some context input c and an output length L, and outputs a key k'.

In this lab, we will be using a key derivation function KDF (specifically, HKDF) to derive the symmetric keying material for use in the MAC and the AEAD scheme, and also to output shared symmetric keys.

- $(\mathsf{sk}, \mathsf{vk}) \leftarrow_R \mathsf{SIG.KeyGen}(\lambda)$: denotes the key generation of a digital signature scheme SIG. It takes as input a security parameter λ, and outputs a signing key sk and a verification key vk.
- $\sigma \leftarrow_R \mathsf{SIG.Sign}(\mathsf{sk}, \mathsf{msg})$: denotes the signing algorithm of SIG. It takes as input a signing key sk and a message msg, and outputs a signature σ.
- $\{0,1\} \leftarrow \mathsf{SIG.Verify}(\mathsf{vk}, \mathsf{msg}, \sigma)$: denotes the verifying algorithm of SIG. It takes as input a verifying key vk, a message msg, and a signature σ and outputs 0 or 1.

  Again, for correctness we need that for all $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{SIG.KeyGen}(\lambda)$, we have:

  $$\mathsf{SIG.Verify}(\mathsf{vk}, \mathsf{msg}, \mathsf{SIG.Sign}(\mathsf{sk}, \mathsf{msg})) = 1.$$

  In this lab, we will be using ECDSA or RSA signatures to instantiate our digital signature scheme.

- $\tau \leftarrow \mathsf{MAC}(k, \mathsf{msg})$: denotes the message authentication algorithm MAC. It takes as input a symmetric key k and a message msg, and outputs a MAC tag τ.

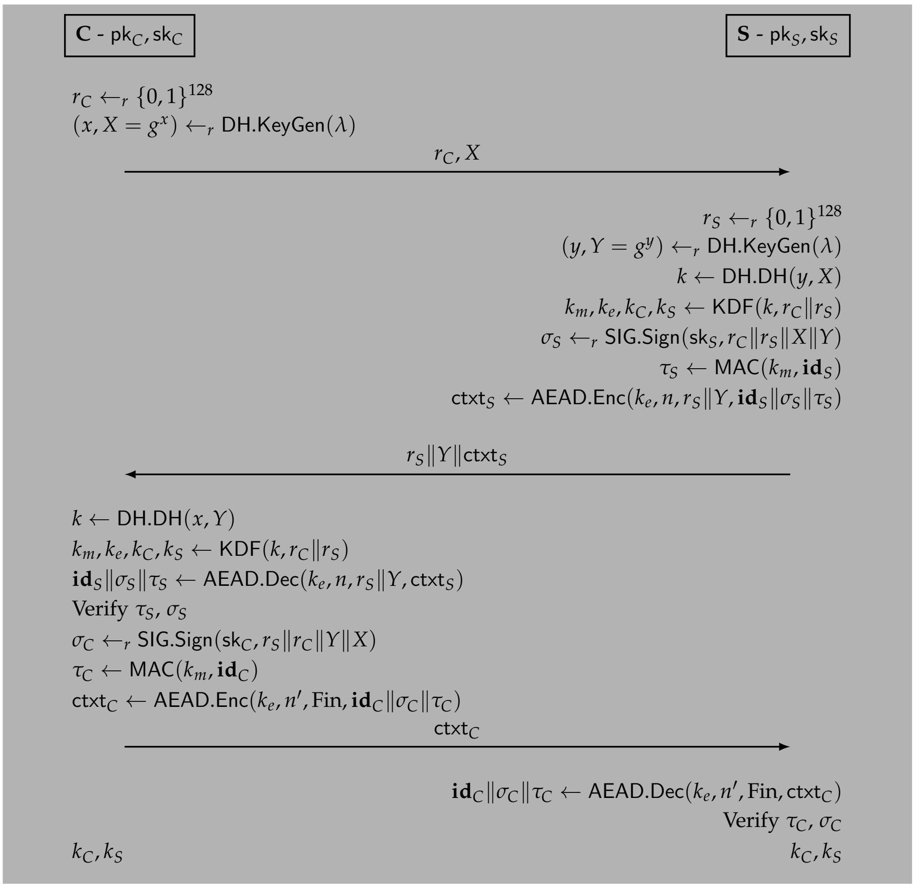
- $k \leftarrow_R \mathsf{AEAD.KeyGen}(\lambda)$: denotes the key generation algorithm of an authenticated encryption scheme with associated data AEAD. It takes as input a security parameter λ and outputs a symmetric key k.
- $\mathsf{ctxt} \leftarrow \mathsf{AEAD.Enc}(k, nonce, ad, \mathsf{msg})$: denotes the encryption algorithm of AEAD. It takes as input a symmetric key k, a nonce $nonce$, some associated data ad and a message msg, and outputs a ciphertext ctxt
- $\{\mathsf{ctxt}, \perp\} \leftarrow_R \mathsf{AEAD.Dec}(k, nonce, ad, \mathsf{ctxt})$: denotes the decryption algorithm of AEAD. It takes as input a symmetric key k, a nonce $nonce$, some associated data ad and a ciphertext ctxt, and outputs either a plaintext message msg or an error symbol $\perp$.

  For correctness we need that for all $k \leftarrow \mathsf{AEAD.KeyGen}(\lambda)$, all nonces $nonce$, for all associated data ad and all messages msg, we have:

  $$\mathsf{msg} = \mathsf{AEAD.Dec}(k, nonce, ad, \mathsf{AEAD.Enc}(k, nonce, ad, \mathsf{msg}))$$

  In this lab, we will be using AES-GCM with 256-bit keys to instantiate our AEAD scheme.

On a high level, we will combine these primitives in the following way:



C - $\mathsf{pk}_C, \mathsf{sk}_C$

S - $\mathsf{pk}_S, \mathsf{sk}_S$

$r_C \leftarrow_r \{0,1\}^{128}$

$(x, X = g^x) \leftarrow_r \mathsf{DH.KeyGen}(\lambda)$

$\xrightarrow{r_C, X}$

$r_S \leftarrow_r \{0,1\}^{128}$

$(y, Y = g^y) \leftarrow_r \mathsf{DH.KeyGen}(\lambda)$

$k \leftarrow \mathsf{DH.DH}(y, X)$

$k_m, k_e, k_C, k_S \leftarrow \mathsf{KDF}(k, r_C \| r_S)$

$\sigma_S \leftarrow_r \mathsf{SIG.Sign}(\mathsf{sk}_S, r_C \| r_S \| X \| Y)$

$\tau_S \leftarrow \mathsf{MAC}(k_m, \mathbf{id}_S)$

$\mathsf{ctxt}_S \leftarrow \mathsf{AEAD.Enc}(k_e, n, r_S \| Y, \mathbf{id}_S \| \sigma_S \| \tau_S)$

$\xleftarrow{r_S \| Y \| \mathsf{ctxt}_S}$

$k \leftarrow \mathsf{DH.DH}(x, Y)$

$k_m, k_e, k_C, k_S \leftarrow \mathsf{KDF}(k, r_C \| r_S)$

$\mathbf{id}_S \| \sigma_S \| \tau_S \leftarrow \mathsf{AEAD.Dec}(k_e, n, r_S \| Y, \mathsf{ctxt}_S)$

Verify τ_S, σ_S

$\sigma_C \leftarrow_r \mathsf{SIG.Sign}(\mathsf{sk}_C, r_S \| r_C \| Y \| X)$

$\tau_C \leftarrow \mathsf{MAC}(k_m, \mathbf{id}_C)$

$\mathsf{ctxt}_C \leftarrow \mathsf{AEAD.Enc}(k_e, n', \mathrm{Fin}, \mathbf{id}_C \| \sigma_C \| \tau_C)$

$\xrightarrow{\mathsf{ctxt}_C}$

$\mathbf{id}_C \| \sigma_C \| \tau_C \leftarrow \mathsf{AEAD.Dec}(k_e, n', \mathrm{Fin}, \mathsf{ctxt}_C)$

Verify τ_C, σ_C

k_C, k_S

k_C, k_S

Skeleton and Implemented Functions

Before we begin, it is worth highlighting some code that has already been provided to you in the skeleton file *sigma.py*. These are essentially functions based on the tinyEC library to implement elliptic curve cryptography [2], and the other core-cryptographic components. For more details about the provided functions, please refer to the appendix at the end of the sheet or consult the skeleton file.

SIGMA

In this lab, you're going to create a class called SIGMA, that will allow a user to initalise a new object sigma_kex. **The skeleton file already contains a SIGMA class template, which you have to complete.** The aim is to allow the user to create the SIGMA protocols described above. We are going to hardcode the concrete instantiations for all cryptographic primitives, see our choices below:

- Diffie-Hellman: ECDH with ECDH_CURVE = "secp256r1"
- Signature Scheme: ECDSA with ECDSA_CURVE = "P-256"

- AEAD scheme: AES-256-GCM
- Hash function: SHA-256
- KDF: HKDF-SHA256
- MAC: HMAC-SHA256

The user should be able to initialise a SIGMA object with a tuple of parameters: (ecdsa_curve, ecdh_curve, identifer, `id_hide_flag`, prng). This will allow the user to specify their preferred elliptic curves for both the Diffie-Hellman and ECDSA algorithms. In addition, this will set the user's identifier **id** used in the SIGMA protocol. This will allow communicating parties to retrieve the pre-distributed public keys pk for their partner to verify the signatures σ exchanged during the protocol execution.

Finally, `id_hide_flag` will allow the user to specify whether they will execute the id-hiding variant of the SIGMA protocol, while `prng` serves as a source for randomness in the protocol.

The SIGMA class should contain the following functions:

- `__init__`: intialises the SIGMA protocol with an ECDSA curve, an ECDH curve, a long-term identifier, a flag that determines whether the protocol does identity hiding, and a pseudorandom number generator (PRNG) that serves as a randomness source for the protocol.
- key_gen: generates and return an ECDSA public key (vk) for the user **id**.
- register_long_term_keys: takes as input `new_id` and `new_pub_key` and adds the public verifying key (in this case, `new_pub_key`) of the user `new_id` to a dictionary, returning no output.
- get_long_term_key: takes as input `pub_id` and retrives the public verifying key of the user `pub_id` from the dictionary, returning the verifying key if present.
- client_init: generates the first SIGMA message consisting of the client nonce, and the ephemeral ECDH value (in bytes).
- server_ecdsa_resp: takes the first SIGMA message as input, and return the second SIGMA message as output, consisting of the server nonce, the ephemeral ECDH value (in bytes) and the server identifier, the server signature, and the server MAC tag. In the identity-hiding variant of SIGMA, it should instead return the server nonce, the server ECDH value (in bytes) and an AEAD ciphertext.
- client_ecdsa_resp: should take the second SIGMA message as input, and return the third SIGMA message as output, as well as k_C, k_S, the output session keys. The third SIGMA message should consist of the client identifier, the client signature and the client MAC tag. In the identity-hiding variant of SIGMA, the third SIGMA message should consist of only a AEAD ciphertext.

- server_ecdsa_fin: should take the third SIGMA message as input, and return k_C, k_S, the output session keys.

The goal is to have the user interact with the API as in the listing that follows:

Listing 1: API description

```
bob_id = convert_id("bob".encode())
alice_id = convert_id("alice".encode())
ID_HIDE_FLAG = False
#ID_HIDE_FLAG = True
alice_sigma = SIGMA(ECDSA_CURVE, ECDH_CURVE, alice_id, ID_HIDE_FLAG)
bob_sigma = SIGMA(ECDSA_CURVE, ECDH_CURVE, bob_id, ID_HIDE_FLAG)
alice_ec_pub_key = alice_sigma.key_gen()
bob_ec_pub_key = bob_sigma.key_gen()
alice_sigma.register_long_term_keys(bob_id, bob_ec_pub_key)
bob_sigma.register_long_term_keys(alice_id, alice_ec_pub_key)
msg = alice_sigma.client_init()
msg_two = bob_sigma.server_ecdsa_resp(msg)
msg_three, client_alice_key, client_bob_key = alice_sigma.client_ecdsa_resp(msg_two)
server_alice_key, server_bob_key = bob_sigma.server_ecdsa_fin(msg_three)
if ((client_alice_key == server_alice_key) and (client_bob_key == server_bob_key)):
        print("Both␣parties␣computed␣the␣same␣key")
else:
        print("Something␣went␣wrong")
```

To summarise, by the end of the lab you should have a python file that completes the following code skeleton:

Listing 2: Code Overview

```
class SIGMA():

def __init__(self, ecdsa_curve, ecdh_curve, identifier,
                                id_hide_flag, prng=CryptoPRNG()):
        Raise NotImplementedError()

# SETUP ALG THAT PRODUCES LONG-TERM SIGNATURE KEYS
def key_gen(self):
        Raise NotImplementedError
        return ec_pub_key

# ADDS (id, vk) PAIRS TO THE PUBLIC KEY DICTIONARY
def register_long_term_keys(self, new_id, new_pub_key):
        Raise NotImplementedError()

# RETRIVES vk FOR id FROM THE PUBLIC KEY DICTIONARY
def get_long_term_key(self, pub_id):
        Raise NotImplementedError()


# INITIALISES A CLIENT STATE, AND PRODUCES THE FIRST
# MESSAGE FROM THE CLIENT TO THE SERVER
def client_init(self):
        Raise NotImplementedError()
        return nonce + x_bytes + y_bytes

# INITIALISES THE SERVER STATE, AND PRODUCES THE FIRST
# MESSAGE FROM THE SERVER TO THE CLIENT
def server_ecdsa_resp(self, client_msg):
        Raise NotImplementedError()
        return msg_two

# VERIFIES AUTHENTICATION MATERIAL USING (id, vk) FROM
# PUBLIC KEY DICTIONARY, PRODUCES THE SECOND MESSAGE
# FROM THE CLIENT TO THE SERVER, OUTPUTS SESSION KEYS
def client_ecdsa_resp(self, server_msg):
        Raise NotImplementedError()
        return msg_three, client_key, server_key

# VERIFIES AUTHENTICATION MATERIAL USING (id, vk) FROM
# PUBLIC KEY DICTIONARY, PRODUCES SESSION KEYS
def server_ecdsa_fin(self, client_msg):
        raise NotImplementedError()
        return self.client_key, self.server_key
```

Some additional functionality you will need to build includes functions that parse the SIGMA messages (in bytes) into valid ECDH values, signatures, MAC tags, etc. that the underlying functions can use. These functions are as follows:

- parse_client_hello(msg, curve) → (client_nonce, client_x_bytes, client_y_bytes, client_ec_pub_key). Note that this will require building another function convert_x_y_bytes_ec_pub which takes byte-representations of the x and y co-ordinates of an EC point, and converts them into a tinyec.point, for which we provide a hint in the listing below.
- parse_ecdsa_server_hello(msg, curve) → (server_nonce, server_id, server_x_bytes, server_y_bytes, server_ec_pub_key, server_signature, server_mac_tag). Similarly to

above, you will need this convert_x_y_bytes_ec_pub function.

- parse_ecdsa_server_hello_id_hide(msg, curve) → (server_nonce, server_x_bytes, server_y_bytes, server_ec_pub_key, server_ctxt). This is the function for parsing the ID hiding version of the server second message.
- parse_ecdsa_ptxt(ptxt) → (identifier, signature, mac_tag). This function will parse the plaintext encrypted in the both the second and third SIGMA id-hiding messages.
- parse_ecdsa_client_resp(msg) → (client_id, client_signature, client_mac_tag). This function will parse the third SIGMA message, in both the normal and id-hiding version.

Listing 3: Code Overview

```
x_bytes = x_int.to_bytes(EC_COORDINATE_LEN, byteorder='big')
y_bytes = y_int.to_bytes(EC_COORDINATE_LEN, byteorder='big')

ec_pub_key = ec.Point(curve, x_int, y_int)
```

Testing your Implementation

We provide a python unit test file *test_sigma.py* to check your implementation. The file contains a set of tests for each of the tasks that follows below. You can run the tests with `python test_sigma.py` or with your favorite testing UI in your IDE. Make sure that the test_sigma.py imports the SIGMA class from your implementation (i.e., check if you change the filename of your implementation). The tests assume that the `SIGMA` class data attributes have the exact name as described in this sheet, and that all the randomness in the protocol is generated by the supplied `PRNG`.

Task-1: Initialisation, Key Generation and Public-Key Registration

To begin, create a function `__init__(self, ecdsa_curve, ecdh_curve, identifier, id_hide_flag, prng=CryptoPRNG())` within the SIGMA class. This will initialise the sigma_kex object that will produce SIGMA messages and output session keys. The state maintained by the SIGMA object is significantly more complex than we have seen in the previous modules. **The object itself must maintain the following variables**:

- *self*.id
- *self*.ecdsa_curve
- *self*.ecdh_curve
- *self*.pub_key_dict
- *self*.id_hide_flag
- *self*.prng
- *self*.ecdsa_sec_key
- *self*.nonce
- *self*.eph_x
- *self*.eph_y
- *self*.eph_sec_key
- *self*.mac_key
- *self*.client_key
- *self*.server_key
- *self*.client_sig_msg
- *self*.enc_key

Setting the first six variables is straightforward: five variables are set to the respective value taken from the input, *self*.ecdsa_curve ← ecdsa_curve, *self*.ecdh_curve←ec_setup(curve_name), *self*.id ← identifier, and *self*.prng ← prng. Additionally, when initialised, the public key dictionary will be empty, and thus: *self*.pub_key_dict = {}. These will all be initialised in the `__init__` function.

The *self*.ecdsa_sec_key will be set by the key generation key_gen algorithm that establishes long-term signing keys. In addition, register_long_term_keys(self, new_id, new_pub_key) will add (**id**, vk) pairs to the public key dictionary as follows:

```
pub_key_dict[new_id] = new_pub_key
```

If you wish to retrieve a public key associated with a given id, do so as follows:

```
server_ec_pub_key = self.pub_key_dict[id]
```

But what about the rest of the values? Well, some of these will only be maintained by objects that are used as clients or servers. The client will need to maintain:

- *self*.nonce
- *self*.eph_x
- *self*.eph_y
- *self*.eph_sec_key

and will set these variables during execution of client_init. Note that the client will need to maintain these values (as internal state) in order to:

1. compute the keys k_e, k_m, k_C, k_S during execution of client_ecdsa_resp.
2. verify the MAC tag and signature sent by the server in the second SIGMA message.
3. compute its own signature over the client nonce r_C, the server nonce r_S, and the client and server public ECDH values.

In comparison, the server will need to maintain:

- *self*.mac_key
- *self*.client_key
- *self*.server_key
- *self*.client_sig_msg
- *self*.enc_key

as it will compute these values during the execution of server_ecdsa_resp. Obviously, it will need to hold onto client_key and server_key in order to be able to output these values at the end of the protocol execution. In addition, it will need to maintain mac_key and client_sig_msg to verify the client MAC tag and signature sent by the client in the third SIGMA message. Finally, if the protocol being executed is the id-hiding variant of SIGMA, it will need to maintain enc_key to decrypt the client message.

We have now addressed how to build `__init__`, key_gen, and register_long_term_keys. So let's move onto the first function that will run when executing the SIGMA protocol.

Task-2: Client Init

Next, we are going to create a function `client_init` that takes only *self* as input, and outputs the first SIGMA message, consisting of `client_nonce` and X, where X is a byte representation of the public ECDH value. Straightforwardly, the function should generate a random nonce (of NONCE_LEN bytes), generate ECDH public/secret values, convert the public value to bytes and maintain the ECDH secret value, nonce and ephemeral ECDH public value in state.

Listing 4: Client Init Function

```
def client_init(self):
    # GENERATE A RANDOM NONCE VIA self.prng.get_random_bytes
    # GENERATE ECDH (sec_key, pub_key) via ec_key_gen(self.ecdh_curve, self.prng)
    # CONVERT ECDH pub_key TO BYTES VIA convert_ec_pub_bytes(eph_pub_key)
    # SET self.nonce
    # SET self.eph_x
    # SET self.eph_y
    # SET self.eph_sec_key
    raise NotImplementedError()
return nonce + x_bytes + y_bytes
```

Now we can move onto the second function that will run when executing the SIGMA protocol.

Task-3: Server Response

At a high-level, the `server_ecdsa_resp` function, which is shown in Listing 5, should take a client SIGMA message, and parse the values correctly. The server will generate a random nonce and ephemeral ECDH values, and perform an ECDH computation using the maintained `eph_sec_key` and the ECDH message sent in the client SIGMA message.

The output of this computation is then used to generate `mac_key`, `client_key`, `server_key` (and, in the id-hiding variant, `enc_key`). These keys will need to be maintained in state to output session keys later, as well as decrypt client ciphertexts and verify client MAC tags.

At this point, the server will use its ECDSA signing key (*self*.`ecdsa_sec_key`) to create a signature over the client and server nonce values, and the client and server ECDH public values. Since the server will need to verify a client signature later, it will need to maintain all these values in *self*.`client_sig_msg` for later. Similarly, the server will use *self*.`mac_key` to generate a MAC tag over its identifier *self*.`id`.

If the id-hiding variant of SIGMA is used, then the server will encrypt the concatenation of its identifier *self*.`id`, the server signature and the server MAC tag, using the server nonce and ephemeral public ECDH value Y as the associated data field, and will output the resulting ciphertext along with the server nonce and the ephemeral public ECDH value Y. Otherwise, the server simply outputs the concatenation of the nonce, identifier, ECDH public value, the server signature and the server MAC tag.

Listing 5: Server Response

```
def server_ecdsa_resp(self, client_msg):
        # PARSE THE CLIENT MESSAGE TO THE FOLLOWING VALUES:
        #    client_nonce, client_x_bytes, client_y_bytes, client_ec_pub_key
        # GENERATE A RANDOM NONCE VIA self.prng.get_random_bytes
        # GENERATE ECDH (sec_key, pub_key) via ec_key_gen(self.ecdh_curve, self.prng)
        # CONVERT ECDH pub_key TO BYTES VIA convert_ec_pub_bytes(eph_pub_key)
        # PERFORM A ECDH COMPUTATION VIA ec_dh(eph_sec_key, client_ec_pub_key)
        # CONVERT THE SECRECT EC POINT TO A BYTE REPRESENTATION VIA
        #    compress(dh_computation).encode()
        # COMPUTE THE MESSAGE THAT THE SIGNATURE WILL BE COMPUTED OVER:
        #    c_nonce, s_nonce, c_ECDH, s_ECDH
        # COMPUTE THE ECDSA SIGNATURE OVER THE MESSAGE USING
        #    ecdsa_sign(self.ecdsa_sec_key, sig_msg, self.prng)
        if (self.id_hide_flag == True):
                # COMPUTE mac_key, enc_key, client_key, server_key USING HKDF
                # SET self.enc_key
        if (self.id_hide_flag == False):
                # COMPUTE mac_key, client_key, server_key VIA HKDF
        # COMPUTE THE MAC TAG OVER self.id
        # SET self.mac_key
        # SET self.client_key
        # SET self.server_key
        # COMPUTE THE CLIENT MESSAGE THAT THE CLIENT WILL SIGN
        # SET self.client_sig_msg
        if (self.id_hide_flag == True):
                # GENERATE THE PLAINTEXT THAT WILL BE ENCRYPTED
                # ENCRYPT THE PLAINTEXT USING AES_GCM
                # GENERATE THE SERVER SIGMA MESSAGE msg_two
        if (self.id_hide_flag == False):
                # GENERATE THE SERVER SIGMA MESSAGE msg_two
        Raise NotImplementedError()
        return msg_two
```

To encapsulate the parsing of the client hello, you will implement a parsing function outside the `SIGMA` class, `parse_client_hello`, which you can then call from `server_ecdsa_resp`. It should parse the first SIGMA message into client_nonce, client_x_bytes, client_y_bytes and client_ec_pub_key, see the listing below. At the beginning of the skeleton file, we have given the exact byte-length of each of the values in the SIGMA messages to help you.

Listing 6: Necessary Sub Functions

```
def parse_client_hello(msg, curve):
        # RECOVER client_nonce from msg
        # RECOVER client_x_bytes from msg
        # RECOVER client_y_bytes from msg
        # CONVERT X_BYTES, Y_BYTES TO client_ec_pub_key
        raise NotImplementedError()
        return client_nonce, client_x_bytes, client_y_bytes, client_ec_pub_key
```

And with that, we can continue onto:

Task-4: Client Response

At a high-level, the `client_ecdsa_resp` function should take a server SIGMA message, and parse the values correctly. The client will perform an ECDH computation using the maintained *self*.`eph_sec_key` and the ECDH message sent in the server SIGMA message, using the output of this computation to generate `mac_key`, `client_key`, `server_key` (and, in the id-hiding variant, `enc_key`).

At this point, the client should verify the signature and MAC tag sent by the server. Note that if the id-hiding variant of SIGMA is chosen, first these values must be recovered by decrypting the ciphertext and parsing the underlying plaintext appropriately. Using the provided identifier, the client should find the public-key stored in the public-key dictionary and use it to verify the signature, and then use `mac_key` to verify the MAC over the identifier.

Afterwards, the client will use its ECDSA signing key (*self*.`ecdsa_sec_key`) to create a signature over the client and server nonce values, and the client and server ECDH public values (hence, the need to maintain *self*.`nonce`, and *self*.`eph_x`, *self*.`eph_y`). Similarly, the client will use `mac_key` to generate a MAC tag over its identifier *self*.`id`.

If the id-hiding variant of SIGMA is used, then the client will encrypt the concatenation of its identifier *self*.`id`, the client signature and the client MAC tag, and will output the resulting ciphertext. Otherwise, the client simply outputs the concatenation of the identifier, the client signature and the client MAC tag.

Listing 7: Client Response

```
def client_ecdsa_resp(self, server_msg):
        if (self.id_hide_flag == True):
                # PARSE THE CLIENT MESSAGE TO THE FOLLOWING VALUES:
                #        server_nonce, server_x_bytes,
                #        server_y_bytes, server_eph_pub_key, server_ctxt
        if (self.id_hide_flag == False):
                # PARSE THE CLIENT MESSAGE TO THE FOLLOWING VALUES:
                #        server_nonce, server_id, server_x_bytes,
                #        server_y_bytes, server_eph_pub_key, server_signature, server_mac_tag
        # PERFORM A ECDH COMPUTATION VIA ec_dh(eph_sec_key, server_ec_pub_key)
        # CONVERT THE SECRECT EC POINT TO BYTES VIA
        #        compress(dh_computation).encode()
        if (self.id_hide_flag == True):
                # COMPUTE mac_key, enc_key, client_key, server_key USING HKDF
                # DECRYPT THE PLAINTEXT WITH aes_gcm_dec(..., ...,
                #        server_nonce + server_x_bytes + server_y_bytes)
                # PARSE THE PLAINTEXT TO: server_id, server_signature, server_mac_tag
        if (self.id_hide_flag == False):
                # COMPUTE mac_key, client_key, server_key USING HKDF
                # RECOVER THE SERVER ECDSA PUBLIC KEY FROM DICTIONARY USING server_id
                # COMPUTE THE MESSAGE THAT THE SIGNATURE WAS BE COMPUTED OVER:
                #        c_nonce, s_nonce, c_ECDH, s_ECDHserver_sig_msg
        # VERIFY THE SIGNATURE WITH ecdsa_verify(...)
        if (result == False):
                raise SigError("Signature␣Verification␣Failed!")
        # GENERATE MAC TAG OVER server_id
        if (mac_tag_prime != server_mac_tag):
                raise MacError("MAC␣Verification␣Failed!")
        # COMPUTE THE MESSAGE THAT THE SIGNATURE WILL BE COMPUTED OVER:
        #        s_nonce, c_nonce, s_ECDH c_ECDH
        # COMPUTE THE SIGNATURE USING ecdsa_sign(self.ecdsa_sec_key, sig_msg, self.prng)
        # GENERATE MAC TAG OVER self.id
        if (self.id_hide_flag == True):
                # GENERATE PLAINTEXT
                # ENCRYPT PLAINTEXT USING aes_gcm_enc(enc_key, ptxt,
                #        "Finished".encode(), self.prng)
                # SET msg_three = ctxt
        if (self.id_hide_flag == False):
                # SET msg_three = id + ec_signature + mac_tag
        raise NotImplementedError()
        return msg_three, client_key, server_key
```

To do so, you will implement several parsing functions outside the SIGMA class:

1. `parse_ecdsa_server_hello`, to parse the second SIGMA message into server_nonce, server_id, server_x_bytes, server_y_bytes, server_ec_pub_key, server_signature, server_mac_tag, see the listing below.

   For the SIGMA id-hiding variant, you will also need the following functions:

2. `parse_ecdsa_server_hello_id_hide`, to parse the second SIGMA message into server_nonce, server_x_bytes, server_y_bytes, server_ec_pub_key, server_ctxt, see the listing below.

3. `parse_ecdsa_ptxt`, which parses the plaintext recovered after decrypting the AEAD ciphertexts into identifier, signature, mac_tag. Thankfully, you will be able to re-use this sub-function in both the Client Response and Server Finished functions.

Listing 8: Necessary Sub Functions

```
def parse_ecdsa_server_hello(msg, curve):
        # RECOVER server_nonce from msg
        # RECOVER server_id from msg
        # RECOVER server_x_bytes from msg
        # RECOVER server_y_bytes from msg
        # RECOVER server_signature from msg
        # RECOVER server_mac_tag from msg
        # CONVERT X_BYTES, Y_BYTES TO server_ec_pub_key
        raise NotImplementedError()
        return server_nonce, server_id, server_x_bytes, server_y_bytes,
                 server_ec_pub_key, server_signature, server_mac_tag

def parse_ecdsa_server_hello_id_hide(msg, curve):
        # RECOVER server_nonce from msg
        # RECOVER server_x_bytes from msg
        # RECOVER server_y_bytes from msg
        # CONVERT X_BYTES, Y_BYTE TO server_ec_pub_key
        # RECOVER server_ctxt from msg
        raise NotImplementedError()
        return server_nonce, server_x_bytes, server_y_bytes,
                server_ec_pub_key, server_ctxt

def parse_ecdsa_ptxt(ptxt):
        # RECOVER identifier from ptxt
        # RECOVER signature from ptxt
        # RECOVER mac_tag from ptxt
        raise NotImplementedError()
return identifier, signature, mac_tag
```

Task-5: Server Finished

The final function `server_ecdsa_fin` should take as input a client message client_msg, and return either session keys k_C, k_S or an error message if τ or σ fails to verify.

Listing 9: Server Finished

```
def server_ecdsa_fin(self, client_msg):
        if (self.id_hide_flag == True):
                # DECRYPT MESSAGE VIA aes_gcm_dec(..., ..., "Finished".encode())
                # PARSE THE PLAINTEXT VIA INTO
                #        client_id, client_signature, client_mac_tag
        if (self.id_hide_flag == False):
                # PARSE THE MESSAGE INTO
                #        client_id, client_signature, client_mac_tag
        # RECOVER THE PUBLIC KEY FROM DICTIONARY USING client_id
        # VERIFY THE SIGNATURE WITH ecdsa_verify(...)
        if (result == False):
                raise SigError("Signature␣Verification␣Failed!")
        # GENERATE MAC TAG OVER client_id
        if (mac_tag_prime != client_mac_tag):
                raise MacError("MAC␣Verification␣Failed!")
        raise NotImplementedError()
        return self.client_key, self.server_key
```

To do this, you will need to implement the following parsing function:

Listing 10: Necessary Sub Functions

```
def parse_ecdsa_client_resp(msg):
        # RECOVER client_id from msg
        # RECOVER client_signature from msg
        # RECOVER client_mac_tag from msg
        return client_id , client_signature , client_mac_tag
```

Once you have implemented these, verify that the keys that the client and server produces are the same.

Evaluation

We will evaluate your sigma submission using exactly the same tests that are provided in the handout. However, we will use private input vectors that will not be made public prior to submission.

Summary of Evaluation Criteria. To summarize, you will be evaluated based on the correctness of your implementation of the SIGMA class. The following points are awarded for correct methods in the SIGMA class based on the provided test cases:

1. `key_gen` (1 points)
2. `client_init` (1 points)
3. `server_ecdsa_resp` (3 points)
4. `client_ecdsa_resp` (3 points)
5. `server_ecdsa_fin` (2 points)

So a total of 10 points is available for this week's part I lab.

Submission Format

Your completed submission for part I this week should consist of a *single* Python file, and should be named "`sigma.py`".

You are expected to upload your submission to Moodle. The submissions for weeks 4 (part 1 & 2) and 5 should be bundled into a single archive file named "module_2_submission_[insert LegiNo].zip". In summary, this file should contain the following files:

- "`sigma.py`", your implementation of the SIGMA protocol from the week 4 lab part I.
- "`tls_crypto.py`", your implementation of the cryptographic functions used by TLS 1.3 from the week 4 lab part II.
- "`tls_handshake.py`", your implementation of the TLS 1.3 Handshake from the week 5 lab.

- "`tls_psk_handshake.py`", your implementation of the PSK functionality of TLS 1.3 from the week 5 lab.
- "`tls_psk_state_machines.py`", your implementation of the TLS 1.3 state machine with PSK and 0-RTT support from the week 5 lab.

In conclusion, *happy coding*!

References

1. Hugo Krawczyk. (2003) "SIGMA: the 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and its Use in the IKE Protocols"
`https://www.iacr.org/cryptodb/archive/2003/CRYPTO/1495/1495.pdf`.

Appendix: Additional Listings

Listing 11: Pseudorandom number generator

```
from tinyec import registry
from Cryptodome.Cipher import AES
from Cryptodome.Protocol.KDF import HKDF
from Cryptodome.Hash import HMAC, SHA256, SHA512
from Cryptodome.Random import get_random_bytes
import math
import secrets

class PRNG:
        def randbelow(self, number: int) -> int:
                """Return a random int in the range [0, n)."""
                pass

        def get_random_bytes(self, nbytes: int) -> bytes:
                """Return a random byte string containing *nbytes* bytes."""
                pass

class CryptoPRNG(PRNG):
        def randbelow(self, number: int) -> int:
                return secrets.randbelow(number)

        def get_random_bytes(self, nbytes: int) -> bytes:
                return secrets.token_bytes(nbytes)
```

Listing 12: TinyEC and Elliptic Curve Cryptography

```
def compress(pubKey):
        return hex(pubKey.x) + hex(pubKey.y % 2)[2:]

def ec_setup(curve_name):
        curve = registry.get_curve(curve_name)
        return curve

def ec_key_gen(curve, prng):
        sec_key = prng.randbelow(curve.field.n)
        pub_key = sec_key * curve.g
        return (sec_key, pub_key)

def ec_dh(sec_key, pub_key):
        shared_key = sec_key * pub_key
        return shared_key
```

As you can see, a lot of the details of the underlying elliptic curve operations are hidden at this level. But you can still glean a high-level understanding of how elliptic-curve Diffie-Hellman key-exchange works, and compare it with the traditional variant of Diffie-Hellman key-exchange.

For key generation, an integer d is randomly sampled from $\mathbb{Z}_n$, where n (seen in Listing 1 as curve.field.n) is order of the point g (the point g generates the group of points on the elliptic curve). The integer d serves as the secret key sk in ec_key_gen. Then, the point g is added to itself d times to compute the public key pk in ec_key_gen. Computing an ECDH shared secret is again scalar multiplication and can be interpreted simply as adding the public-key pk to itself sk-many times.

Listing 13: ECDSA Signatures

```
def ecdsa_key_gen(curve_name, prng):
        key = ECC.generate(curve=curve_name, randfunc=prng.get_random_bytes)
        pub_key = key.public_key()
        pub_key_pem = pub_key.export_key(format='PEM')
        return (key, pub_key_pem)

def ecdsa_sign(sec_key, msg, prng):
        h = SHA256.new(msg)
        sig = DSS.new(sec_key, 'fips-186-3', randfunc=prng.get_random_bytes)
        signature  = sig.sign(h)
        return signature

def ecdsa_verify(pub_key_pem, msg, signature):
        pub_key = ECC.import_key(pub_key_pem)
        h = SHA256.new(msg)
        sig = DSS.new(pub_key, 'fips-186-3')
        try:
                sig.verify(h, signature)
                result = True
        except ValueError:
                result = False
        return result
```

Listing 14: HMAC

```
def gen_mac(key, msg):
        hmac = HMAC.new(key, digestmod=SHA256)
        hmac.update(msg)
        tau = hmac.digest()
        return tau
```

Listing 15: HKDF

```
key_one, key_two, key_three = HKDF(seed, 32, salt, SHA256, 3)
key_one, key_two, key_three, key_four = HKDF(seed, 32, salt, SHA256, 4)
```

Listing 16: AESGCM Encryption and Decryption

```
def find_ptxt_pad_len(ptxt):
        message_len = len(ptxt)
        pad_len = BLOCK_LEN - (message_len % BLOCK_LEN)
        if (pad_len == 0):
                pad_len = BLOCK_LEN
        return pad_len

def add_ptxt_padding(ptxt):
        ptxt_set = []
        for i in range(len(ptxt)):
                ptxt_set.append(ptxt[i])
        pad_len = find_ptxt_pad_len(ptxt)
        for i in range(pad_len):
                ptxt_set.append(pad_len)
        pad_ptxt_bytes = bytes(ptxt_set)
        return pad_ptxt_bytes

def remove_ptxt_padding(ptxt):
        pad_ptxt_len = len(ptxt)
        pad_len = ord(chr(ptxt[pad_ptxt_len-1]))
        ptxt_len = pad_ptxt_len - pad_len
        plaintext = ptxt[:ptxt_len]
        return plaintext

def aes_gcm_enc(key, msg, ad, prng):
        nonce = prng.get_random_bytes(BLOCK_LEN)
        cipher = AES.new(key, AES.MODE_GCM, nonce=nonce)
        cipher.update(ad)
        pad_msg = add_ptxt_padding(msg)
        ctxt, tag = cipher.encrypt_and_digest(pad_msg)
        nonce_ctxt_tag = nonce + tag + ctxt
        return nonce_ctxt_tag

def aes_gcm_dec(key, ctxt, ad):
        nonce = ctxt[:BLOCK_LEN]
        tag = ctxt[BLOCK_LEN:2*BLOCK_LEN]
        ciphertext = ctxt[2*BLOCK_LEN:]
        cipher = AES.new(key, AES.MODE_GCM, nonce)
        cipher.update(ad)
        pad_ptxt = cipher.decrypt_and_verify(ciphertext, tag)
        ptxt = remove_ptxt_padding(pad_ptxt)
        return ptxt
```